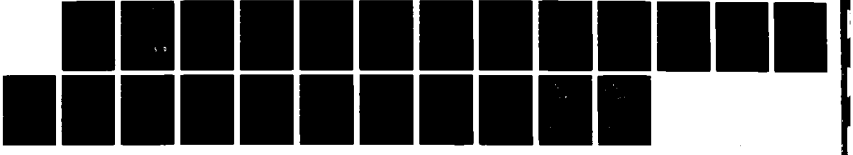
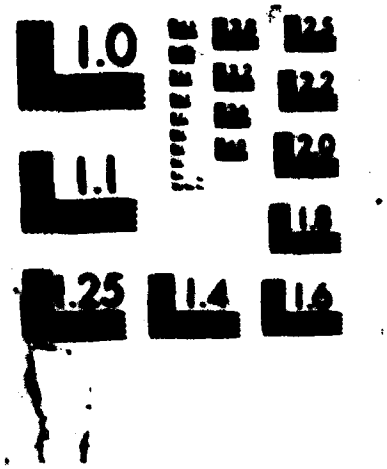
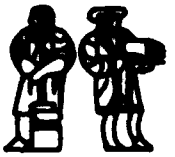
AD-A187 011 APPROXIMATING THE SIZE OF A DYNAMICALLY GROWING ASYNCHRONOUS DISTRIBUTED..(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENEE.. 1/1

UNCLASSIFIED B AWERBUCH ET AL. APR 87 MIT/LCS/TM-320 F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
2.0
2.2
2.5

LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-328

APPROXIMATING THE SIZE OF A DYNAMICALLY GROWING ASYNCHRONOUS DISTRIBUTED NETWORK

Baruch Awerbuch

Serge A. Plotkin

April 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

AD-A187011

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-328	5 MONITORING ORGANIZATION REPORT NUMBER(S) N00014-80-C-0622
6a NAME OF PERFORMING ORGANIZATION Laboratory for Computer Science 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Approximating the Size of a Dynamically Growing Asynchronous Distributed Network

12 PERSONAL AUTHOR(S)
Awerbuch, Baruch and Plotkin, Serge A.

13a TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM _____ TO _____	April 1987	17

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	termination detection, leader election, distributed networks

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We show how to approximate up to a constant factor the size of a dynamically growing asynchronous distributed network. The technique presented in this paper has an amortized message complexity of $O(\log^2|V|)$ per node, where $|V|$ is the final size of the network.

This technique appears to be a useful primitive in constructing distributed algorithms. In this paper we show how to apply this technique to construct an efficient distributed algorithm for leader election in a faulty network. The algorithm has a message complexity $O(|E| + |V|\log^2|V|)$, which is an improvement of $O(\log^2|V|)$ over the previously known algorithms.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE (Include Area Code) (617) 253-5894 / 22c OFFICE SYMBOL

Approximating the Size of a Dynamically Growing Asynchronous Distributed Network

Baruch Awerbuch* Serge A. Plotkin†

Laboratory for Computer Science
MIT
Cambridge MA 02139

*Supported by the Air Force contract TNDGAFOSR-86-0078
†Supported by the Advanced Research Projects Agency of the Department of Defense under the contract N00014-80-C-0622.

Abstract

We show how to approximate up to a constant factor the size of a dynamically growing asynchronous distributed network. The technique presented in this paper has an amortized message complexity of $O(\log^2 |V|)$ per node, where $|V|$ is the final size of the network.

This technique appears to be a useful primitive in constructing distributed algorithms. In this paper we show how to apply this technique to construct an efficient distributed algorithm for leader election in a faulty network. The algorithm has a message complexity $O(|E| + |V| \log^2 |V|)$, which is an improvement of $O(\log^2 |V|)$ over the previously known algorithms.

Keywords: termination detection, leader election, distributed networks.

1 Introduction

A situation that is often encountered in distributed algorithms is that the subnetwork of nodes participating in the algorithm is dynamically growing. In some cases new node is added to the subnetwork after being activated by a message from a node that is already in the subnetwork. In other cases a node is activated by some asynchronous external event. In both cases it seems advantageous to be able to approximate the size of the active subnetwork. A natural example is when we want to terminate the algorithm when the subnetwork reaches certain size. Another example is when we want to start a new iteration of some algorithm each time the subnetwork grows by a constant factor.

In this paper we define the approximate counting problem which abstracts some of the difficulties encountered in maintaining dynamically growing networks. Informally, the approximate counting problem is the problem of maintaining a constant factor approximation to the size of the network. In this paper we show how to solve this problem with an amortized message complexity of $O(\log^2 |V|)$ per node, where $|V|$ is the final size of the network.

A problem similar to the approximate counting problem was independently introduced by Afek and Saks [AS87]. Using their techniques it is possible to solve the approximate counting problem with amortized message complexity of $O(\log^3 |V|)$

per node, which is worse by a factor of $\log |V|$ compared to our technique.

In this paper we show how to apply the approximate counting to efficiently solve the problem of a leader election in a faulty network. The problem of finding a leader is one of the fundamental problems in communication networks. It is an important tool for breaking symmetry between processors executing the same algorithm and allows application of highly centralized protocols in a completely decentralized environment. We consider a problem of electing a leader in a faulty network. The links of the network are assumed to be either up or down throughout the execution of the algorithm. Without this assumption, the consensus in a faulty asynchronous network is impossible, as it was shown in [FLP82]. As proved in [FL84], $\Omega(|E| + |V| \log |V|)$ messages are required for leader election in such a network. A restricted case of leader-election on a ring was treated in [GS86], who give a $O(|E| + |V| \log |V|)$ algorithm. The leader-election in general networks was addressed in [BK86], who give an $O(|V|^2)$ algorithm, and in [AS87].

In this paper we show how to apply the approximate counting technique to construct an algorithm for leader election in a faulty network with message complexity of $O(|E| + |V| \log^2 |V|)$. We use a slightly weaker definition of the leader election problem, compared to the definition used by Afek and Saks [AS87], who give an $O(|E|+|V| \log^5 |V|)$ message complexity leader election algorithm. Their algorithm, adapted for our definition of the leader election algorithm, has message complexity of $O(|E| + |V| \log^4 |V|)$.

Using our techniques together with the techniques introduced in [AS87], it is possible to achieve an $O(|E| + |V| \log^3 |V|)$ message complexity algorithm for the stronger notion of the leader election problem [AAPS87].

This paper is organized as follows. First, we describe the model and give complexity measures. In section 3 we present the main technique that allows us to solve the approximate counting problem on a dynamically growing tree. In section 4 we describe how to use the approximate counting on a tree to elect a leader in a network with faulty edges. Section 5 present a more detailed description of the

leader-election algorithm. The correctness and the complexity of the leader election algorithm are addressed in sections 6 and 7.

2 The Model and Complexity Measures

We consider an asynchronous communication network [Awe85]. This is a point-to-point (or store-and-forward) communication network, described by an undirected *communication graph* $G = (V, E)$, where the set of nodes V represents processors of the network and the set of edges E represents bidirectional non-interfering communication channels operating between neighboring nodes. No common memory is shared by the node's processors and there is no notion of a global clock.

The following complexity measures are used to evaluate the performance of distributed algorithms. The *Communication Complexity* (C), is the worst case total number of elementary messages sent during the algorithm, where an elementary message contains at most $O(\log |V|)$ bits. The *Time Complexity* (T), is the maximum possible number of time units from the start to the completion of the algorithm, assuming that the inter-message delay and the propagation delay of an edge are at most one time unit. This assumptions are used only for the purpose of evaluating the performance of the algorithm and can not be used to prove its correctness, since the algorithm is event-driven.

3 Approximate Counting on a Tree

This section defines the *approximate counting on a tree* (ACT) problem and presents an algorithm to solve it with amortized message complexity of $O(\log^2 |V|)$ per node. The technique described in this section is the main building block for the leader-election algorithm described in the subsequent sections.

The approximate counting on a tree problem is defined as follows. Assume that in the beginning we have a network with no edges. New edges are asynchronously

added to the network in such way that at every point in time the network consists of a single tree and some isolated nodes. (By saying that a new edge is added we mean that the endpoints are informed about the existence of this edge.) We say that the *approximate counting on a tree* (ACT) problem is solved if for every final tree of size $|V|$, the root of this tree will eventually know the value of $|V|$ within a given constant factor. In other words, the root of the tree will eventually come to a conclusion that the size of the tree is at least $|V'|$, where $1 \geq |V'|/|V| \geq c$, for some given constant c.

A straightforward solution to this problem is to communicate with the root each time a new node is added to the tree. Though this solution has an optimal time complexity of $O(|V|)$, the message complexity is very large – $O(|V|^2)$. The main idea behind our algorithm is to trade off time for messages by delaying communication with the root until 'enough' new nodes were added to the tree, and charge the communication cost on the *new* nodes.

We will describe the algorithm in terms of *waves of messages*. Consider a set S of connected nodes and assume there exists a spanning tree T_s of S with root r. The wave is a special message generated by this root. It propagates through the tree T_s until it reaches the leaves. A leaf returns this message to the node it received it from. Each non-leaf node waits till it has received the message back from all the nodes it has sent it to, and returns the message to the node it has received it from. Conceptually, the wave *sweeps* the nodes in S. Note, that a wave will eventually return to the root T_s. New wave can be generated only after the first one has returned back to the initiating node. In the following description, we use the Freeze wave. Any node that receives this wave suspends responses to any message that comes from a node outside of S. Such message is saved in a special queue and answered only after the Defreeze wave sweeps this node.

To make the presentation of the algorithm more uniform, we introduce the notion of *cluster*. Cluster is a set of connected nodes. The node that is closest to the root of the tree is the *cluster-root*, which governs the behavior of the nodes in the cluster. A cluster-root knows *exactly* the size of its cluster. Naturally, in the beginning, each node is a cluster-root of a cluster of size 1.

We have assumed that at any point the network consists of a tree and some isolated nodes. Therefore each new edge connects the root of one cluster to some node in another cluster. In this case we say that the first cluster is ***directly hooked*** onto the second one. To simplify the description of the algorithm we introduce the notion of ***approximation property***. We say that this property is violated at some cluster if the total size of the clusters directly hooked onto it is more than a constant factor larger than its own size.

The main part of the algorithm is to detect the violation of the approximation property inside every cluster. It is clear that if we will sweep a cluster, counting the size of all the clusters directly hooked onto it, each time the cluster-root will discover that the approximation property is violated, any node will participate in at most $O(\log |V|)$ sweeps, where $|V|$ is the final size of the network.

To facilitate the propagation of information inside a cluster, we maintain the following data structure. This data structure is initialized each time we sweep the cluster with Count wave after the cluster-root has detected violation of the approximation property. Denote the set of nodes in the cluster and in all the clusters directly hooked onto it by S. Assume $2^{i-1} \leq |S| \leq 2^i$. The nodes in S are divided into at most 2 disjoint subsets of size at least 2^{i-1}, at most 4 disjoint subsets of size at least 2^{i-2}, e.t.c., where the node of a subset of size 2^j that is the closest to the cluster-root, will be denoted by $master_j$. We construct this data-structure when the Count wave sweeps the tree on its way back to the cluster-root.

When node u gets back the reflection of the Count wave from all of its children, it computes the array L_u and sends it to its father v together with the Count message, where $L_u[i]$ is the maximum distance from any one of the descendants of u to $master_i$, for all $1 \leq i \leq \log |V|$. Denote by $\hat{E}$ the edges of the tree. The node v computes

$$L_v[i] = \max\{L_u[i] \mid (u,v) \in \hat{E}\} + 1; \quad 1 \leq i \leq \log |V|.$$

If for some i, $L_v[i] \geq 2^i$, v will be $master_i$.

Note, that each node belongs to at least one and at most $\log |V|$ such subsets; moreover from each node we can reach in at most 2^j steps either the cluster-root

or some $master_j$. Associate with every $master_i$ the closest (up the tree) $master_{i+1}$. The data structure of masters, as defined above, embeds a new tree, which we call the *Collection/Distribution Tree* (CDT) inside the network. The depth of the CDT is $O(\log |V|)$ and each edge between $master_i$ and $master_{i+1}$ has length of at most 2^i in the original tree.

After the construction of CDT, the algorithm proceeds by collecting *reports* about the number of new nodes. We say that a report message carrying information about k new nodes weights k units. Consider a cluster V_u with cluster-root u that is directly hooked onto another cluster V_w. Each time u detects that since it has sent the last report its cluster size has increased at least twice, it sends a new report message with weight that is the increment of its size since the last report. Assume the increment was $|S|$. In this case the report will be sent to $master_{\log|S|+\log\log|V_w|-\log\alpha}$ in CDT of cluster V_w. This report is delayed by this master until another report is received. In general, when $master_i$ gets 2 reports, it replaces them by a report with weight equal to the sum of the weights, and sends it up the CDT to $master_{i+1}$ or to the cluster-root, whichever is closer. To inhibit useless messages, a node will stop forwarding reports after it had forwarded a report with weight equal or larger then the weight of the cluster it belongs to.

Every time a cluster-root receives a report of some weight, it updates the lower bound on the number of nodes in the clusters directly hooked onto it. Every time this lower bound exceeds some constant fraction $\alpha < \frac{1}{6}$ of the size of its cluster, the cluster-root initiates a *cluster-merge* process. This process starts with cluster-root generating a Freeze wave that sweeps all the nodes in the cluster. Then a Count wave is generated by the cluster-root, creating the CDT and counting (exactly) the number of nodes in clusters directly hooked onto the cluster. The cluster-roots of these clusters cease to act like cluster-roots and this effectively merges them together into the cluster that initiated the cluster-merge process. When the Count wave returns to the cluster-root, it generates the Defreeze wave. Nodes receiving this wave resume responding to all the messages.

Remark: The intuition for the above idea is due to Oded Goldreich [Gol86]. Assume

that we have a number of small trees hooked onto a larger one, (where by 'hooked' we mean that there exists an edge from the root of the small tree to some node in the larger one). Let every small tree send a message describing its size into the larger tree, so that this message will be forwarded only number of times proportional to the size of the smaller tree. If the paths of two such messages representing trees of comparable size cross, we "combine" them into a larger message and send it forward as if it was sent by a tree of twice the size. Conceptually, those trees are "paired". Intuitively, in the "steady-state" there are not many messages 'stuck' in the large tree and hence the root of the large tree has a good approximation on the size of the trees hooked onto it.

We claim that if the approximation property is violated, a new Count wave will be eventually generated by the root. Consider some cluster V_r with cluster-root r, size $|V_r|$, lower bound on the size of the clusters directly hooked onto it $|W_r|$, and the actual size of the clusters directly hooked onto it $|W'_r|$. Let R be the root of the tree and denote by $|V|$ the total size of the tree.

Lemma 1 *When the algorithm terminates, the root of the tree has a constant factor approximation on the size of the tree.*

Proof: Assume that from some point on, no new Count wave will be generated. From the description of the algorithm it can be seen that *master*$_i$ can delay a report that weights at most $2^i\alpha/\log|V_r|$. There are at most $|V_r|/2^i$ masters at level i of the CDT. The depth of the CDT is at most $\log|V_r|$ and hence the total weight of reports delayed in the nodes of the CDT is at most $\alpha|V_r|$.

A cluster sends report each time its size increases by a factor of two. Hence, the total weight of reports sent to V_r is at least $\frac{1}{2}|W'_r|$. Therefore the error $|W'_r| - W_r \leq 2\alpha|V_r|$. On the other hand, $W_r < \alpha|V_r|$ because otherwise a new Count wave will be eventually generated. and hence $|W'_r| \leq 3\alpha|V_r|$.

In particular, for the cluster whose cluster root is the tree root R, we have:

$$\begin{aligned} |V| &\leq \sum_{i=0}^{\infty} (3\alpha)^i |V_R| \\ &\leq \frac{1}{1-3\alpha} |V_R| \end{aligned}$$

∎

Now we show that the message complexity of the algorithm is $O(|V| \log^2 |V|)$.

Lemma 2 *The amortized message complexity of the ACT algorithm is* $O(\log^2 |V|)$ *per node, where* $|V|$ *is the final size of the tree.*

Proof: Both **Freeze** and **Defreeze** waves use a constant amortized number of messages per node. The **Count** wave uses $O(\log |V|)$ amortized number of messages per node. Every time the **Count** wave sweep nodes in a cluster, the size of this cluster increases by a constant factor and therefore there are at most $O(|V| \log^2 |V|)$ messages used by these waves.

Consider all the reports sent in the situation when a larger (or equal size) cluster is hooked onto a smaller (or equal size) cluster. Any node propagates only a single message associated with one of these reports per increase by a constant fraction $(1+\alpha)$ in the size of the cluster this node belongs to. Hence, the total cost of these reports is $O(|V| \log |V|)$.

Consider the rest of the reports. Charge the messages used by the report sent by a cluster-root of some cluster onto the nodes whose weight is reported. Consider a report that is sent by $master_i$ to $master_{i+1}$ in some CDT. The cost of such report is at most $\log |V|/\alpha$ amortized over nodes charged for this report. On the other hand, if a node was charged for this report once, the next time it will be charged for a report the size of its cluster will be at least 2^{i+1}.

Hence, for every i, each node is associated with at most a single report from $master_i$ to $master_{i+1}$ in some CDT. There are at most $|V|/2^i$ reports of weight

2^i, $0 \leq i \leq \log |V| + 1$. Therefore reports use $O(\log^2 |V|)$ messages amortized per node in the final tree. ∎

4 Overview of the Leader-Election Algorithm

This section presents an overview of the leader election algorithm. To simplify the presentation, we may view this algorithm as two algorithms being executed in parallel, where the local output of one is the local input to another. The first algorithm constructs a spanning tree of the operational part of the network and the second one maintains an approximation of the size of the tree, terminating the first one when the size of some tree exceeds $\frac{1}{2}|V|$.

We assume that each processor has a unique identification number (ID), represented as an integer $O(\log |V|)$ bits wide. Some of the network links are not operational, i.e. no messages can pass through in either direction. The status of a link (operational or non-operational) is assumed to be fixed throughout the algorithm. We confine ourselves only to event-driven algorithms, which do not use time-outs. Hence, there is no possibility to determine that an edge is non-operational.

This model can be viewed as describing a situation in which a network was damaged and the purpose of the algorithm is reorganize the network and choose a new leader. The requirement that we do not rely on timeouts is reasonable in case the ratio of the largest link-delay to the average link-delay is high.

We define the leader election problem as follows. At any node, the local input to the algorithm consists of a list of adjacent network edges together with $|V|$, the size of the network. Some of the links are not operational and there is no prior knowledge about which of the edges are operational. Some component of size $\frac{1}{2}|V|(1+x)$ $(0 < x < 1)$ of the network is guaranteed to be connected. For simplicity, we assume in the future that all the network is connected, i.e. $x = 1$.

The algorithm is triggered at a different time instances at each node. The output of the algorithm is produced locally at each node, not necessarily at the same time.

It consists of the ID of some node, which we will call the *leader*. All the nodes have to choose the same leader.

To construct the spanning tree we use a variant of the Minimum-Spanning-Tree (MST) algorithm, presented in [GHS83]. This algorithm maintains a forest induced by the edges known to be operational. The algorithm hooks the trees of the forest one onto another, finally creating a single tree. The operation of each tree in the forest is governed by its root.

The hooking operation of one tree on another may be viewed as a creation of a new tree, with its root being the root of the second tree. Each tree is assigned an integer, that is called the *level* and initially equals zero. A tree hooks itself onto another tree only if the level of the second tree is greater or equal to the level of the first tree. If the levels are equal, the level of the newly created tree is the level of the parts plus one. To distinguish this case from the case when a low-level tree hooks itself on a higher-level one, we say that in this case a *core* was created. After two (or more) equal-level trees are hooked together, the root of one of these trees becomes the root of the new tree (we call such a root the *core-root*). The unique identification number of each node is used to break ties.

Informally, when a node detects that it has a link to another tree on at least the same level, it propagates this information to the root of its tree. The root chooses one of these links and initiates a hooking process. After the process is completed, the hooked tree is scanned and all the nodes are informed about the new level of the tree. If a core was created, both trees are scanned, the nodes are informed about the new level, and the number of nodes in the newly created tree is counted.

The above spanning-tree algorithm is a non-terminating one, because a root of a tree can never be sure that there exists no operational link between one of its nodes and a node in another tree that is on at least the same level. Note, that when a low-level tree hooks itself on a higher-level one, we do not inform the higher-level tree about it, and hence the root of the newly created tree does not know the exact number of nodes in its tree. It is easy to see that counting the nodes in a tree after each hooking operation results in an $O(|V|^2)$ message-complexity algorithm. To reduce the number of messages, we should inform the root only when the number

of nodes in the tree increases by at least a constant fraction. Setting this constant fraction to less then $\frac{3}{2}$, we can determine the termination of the algorithm – the root of a tree in which there are at least $\frac{1}{2}|V|$ nodes will declare itself a leader.

In order to maintain an approximation to the size of its tree, each root executes a variant of the approximate counting algorithm. The edges that the ST algorithm has designated to be in the tree are considered to be the input to the approximate counting algorithm. Every time a core is created, the root of the newly created tree restarts the approximate counting algorithm, the nodes in the tree being considered as a single cluster. There is a minor difference between the algorithm executed by each root and the ACT algorithm described in section 3. In ACT algorithm any new edge was between some cluster-root and some other node. Here, when one tree is hooked onto another tree, it corresponds to an edge that appears between two nodes in different trees. The solution is straightforward. Every time one tree is hooked onto another, it is transformed so that the pointers will point in the right direction. Together with this transformation a new CDT is built.

5 Spanning Tree Algorithm

This section presents a more detailed description of the Spanning Tree algorithm, which constructs a spanning tree in a network with faulty edges. This algorithm can be viewed as a variation of the MST algorithm in [GHS83]. The MST algorithm can not be used unchanged because it works only for static networks. An algorithm similar to the one described below was independently discovered by Afek and Saks [AS87]. In order to be able to use our approximate counting algorithm, we need a somewhat different algorithm.

The local input to the ST algorithm consists of a list of adjacent links. Each node starts the algorithm execution by sending one message through each adjacent link. If a message was received through a link, a node assumes this link to be operational. These messages arrive asynchronously and hence a node can never be sure whether it has an operational link in addition to the ones it knows about

already.

The ST algorithm organizes the nodes into trees with tree-edges being the links known so far to be operational. The operation of a tree is governed by its root. The trees are hooked one onto another, finally producing a spanning tree of the operational part of the network. We assume that each node has a distinct ID. The ID of a root is considered to be the ID of its tree and is known to all the nodes in the tree. Later we use the term 'node ID' as a shorthand for 'the ID of the root this node belongs to'.

Initially, each node is a tree of level zero and the nodes are in the Find state. In this state the operation of a node is very similar to its operation in the MST algorithm, except that the Report-Edge message is returned only after an acceptable edge was found. More precisely, each operational link may be in one of 3 states: *Basic*, *Rejected* and *Accepted*. When a link becomes operational, it is in the *Basic* state. In the Find state, each node picks an adjacent *Basic* edge and sends a Test message through it. If a node receives a Test message, it responds as follows:

- If the message is from a node that belongs to the same tree, the Reject message is returned and the edge is marked *Rejected*.
- If the message is from a different tree, and the level of the receiving node is greater than the level of the sending node or the levels are equal but the ID of the receiving node is higher, the Accept message is returned.
- Otherwise, the receiving node does not reply until the preconditions of one of the previous cases are satisfied.

Upon acceptance of a Reject message, the node sends Test through the next *Basic* edge, or waits till such an edge appears if there are no *Basic* edges. Upon acceptance of an Accept message, the node sends Report-Edge message to its father and changes its state to Found. Note that the state of this edge was not changed. Upon acceptance of a Report-Edge message, the node propagates this message to its father, changes its state to Found, and disregards all subsequent Report-Edge messages.

When the root receives a Report-Edge message containing information about a link that leads to a tree with higher level, the root sends a message Take-the-Lead

to the node (the 'hooking node') that discovered this link (the 'hooking edge'), and the *hooking process* is started. First, the hooking node freezes all the nodes in the hooking tree by generating a wave of Freeze messages that propagates through all the nodes reachable from the hooking node through the edges chosen to be in the tree. After acceptance of such message, a node enters the Freeze state and suspends responses to all messages from the outside the frozen tree until further notice. If the wave reaches a node that has received a Report-Edge-Core message since the previous Freeze wave, this information is propagated to its father. When this wave returns to the the 'hooking node', it becomes the new root. If the wave brings information that one of the nodes in the tree has received a Report-Edge-Core message, the level is incremented. The hooking node generates Change-Pointers wave that sweeps the frozen nodes, updates the direction of their pointers and informs them about their new level and ID. Upon receiving the reflection of this wave back, the 'hooking node' generates the Defreeze wave, changing the state of all the previously frozen nodes to Find. The root of the hooked tree ceases to be a root and this completes the hooking operation.

If the root receives information about a link that leads to a tree with the same level but higher ID, it sends the message Take-the Lead to the hooking node u. The node u sends Report-Edge-Core message through the hooking edge to node v in the neighbor tree and designates this edge as one of the edges of the spanning tree. If v is still in the Find state and its level is still equal to the level of the sending node, it sends Report-Edge message to its father and changes its state to Found. If the level of v-s tree is higher than the level of u-s tree, v returns a No-Core message to u and u proceeds to behave exactly like the hooking node during regular hooking operation.

If the root receives information about a link that leads to a tree with the same level but lower ID, it means that there is at least one tree on the same level that hooked itself onto this tree. The root proceeds to generate Freeze, Change-Pointers and Defreeze waves, behaving exactly like the hooking node during the regular hooking operation.

6 Correctness of the Leader Election Algorithm

In this section we show that the leader-election algorithm is correct. First we show that the ST algorithm described in section 5, produces a spanning tree of the operational part of the network. Then we show that the approximate-counting algorithm, described in section 3 detects when the size of a tree whose root is a potential leader exceeds $\frac{1}{2}|V|$, terminating the ST algorithm.

In order to show that the ST algorithm produces a spanning tree, we have to show first that a node will return to the Find state from every other state. A node leaves this state either when it finds a hooking edge, after it receives Report-Edge or Report-Edge-Core message, or when the Freeze wave reaches it. If the Freeze wave was the cause the node left the Find state, it will reenter this state eventually, because the Freeze, Change-Pointers and Defreeze waves propagate independently of every other activity, eventually returning every node on their path to the Find state. Hence, a node will not return to the Find state only if it has left it because it has sent a Report-Edge message and the Freeze wave has never reached it afterwards.

In case a Report-Edge message was sent, one of such Report-Edge messages will reach the root. If the Report-Edge message that reaches the root contains a link to a higher level tree, the Freeze wave will be generated and it will reach all the nodes in the tree. If it contains a link to a node in a tree on the same level, and the Report-Edge-Core reaches this node after its level changes, a regular hooking operation is performed and again the Freeze wave will reach all the nodes in this tree. If the chosen link leads to a node in a tree on the same level, and the Report-Edge-Core reaches this node before its level changes, a Report-Edge message is generated. If this message reaches the root, Freeze wave will be generated, covering all the nodes in the trees.

The only case not covered so far is when the root of tree T_1 has sent Report-Edge-Core message to a node in T_2 and the root of T_2 has sent Report-Edge-Core message to a node in T_3, and so on. By the construction, the Report-Edge-Core message is sent only to nodes in trees with larger ID, and therefore there are no cycle in the created waiting relation. Hence, there exists a tree T_k that its root has received a

Report-Edge message containing information about either a link to a tree on a higher level or a link to a tree on the same level but lower ID. This root will create a Freeze wave that will reach all the nodes in the trees $T_1, T_2, \ldots T_k$.

From the previous discussion it follows that a node will always return eventually to the Find state. If there are two trees connected by an operational link, this link is *Basic*. One of the end-points of this link is in the lower-level tree or in a tree with smaller ID. This node will eventually enter the Find state, will discover this link, and will Report-Edge it to its root. Therefore, as long as there exists an operational link that connects two nodes that belong to two different trees, the algorithm can not deadlock. Hence, eventually, the ST algorithm will find a spanning tree of the network.

7 Complexity Analysis of the Leader Election Algorithm

This section presents the message-complexity analysis of the leader-election algorithm. Like the description of the algorithm, it is convenient to separate the complexities of the ST algorithm and the complexity of the approximate-counting algorithm.

The analysis of the message-complexity for the ST algorithm is similar to the analysis presented in [GHS83]. Each time the level of the tree is incremented by one, the number of the nodes in this tree increases by at least a factor of two. Hence, the number of levels is $O(\log |V|)$.

Every time a tree finds a hooking-edge, its level increases. In the worst case, during this operation a tree edge is used to send the following messages.

1. The Report-Edge message, reporting about an edge which is a candidate to be a hooking edge.
2. Two messages for the propagation of the Freeze wave.
3. Two messages for the propagation of the Change-Pointers wave.

4. Two messages for the propagation of the Defreeze wave.
5. A message Take-the-Lead.
6. A message Report-Edge-Core.

Each operational edge can be rejected only once and each node in a tree can receive at most one Accept message and send at most one Test message between level increments. In addition, because of the cluster-merge process that runs in parallel with the Change-Pointers wave, we use another $O(\log |V|)$ messages per node. Hence, for each node, every time the level of the tree it belongs to increases, we use $O(\log |V|)$ messages in the ST algorithm.

As it was mentioned in section 4, each time a tree is hooked onto another tree, a new CDT is built in the first tree. The amortized cost of building the CDT is $O(\log |V|)$ per participating node. On the other hand, each time a node can participate only in $O(\log |V|)$ hooking operations and hence the total amortised cost for this operation is $O(\log^2 |V|)$.

This concludes the proof that the message complexity of the leader election algorithm is $O(|V| \log^2 |V|)$.

Acknowledgments

Thanks are due to Oded Goldreich for collaboration in the earlier stages of the research.

References

[AAPS87] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. An $O(V \lg^3 V)$ algorithm for leader-election in faulty networks. 1987. (Manuscript in preparation).

[AS87] Y. Afek and M. Saks. Detecting global termination conditions in face of uncertainty. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, 1987. (To appear).

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[BK86] R. Bar-Yehuda and S. Kutten. *Fault Tolerant Leader Election With Termination Detection, in General Undirected Networks.* Technical Report CS-1986-2, Duke University, January 1986.

[FL84] G. Frederickson and N. Lynch. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proceedings of the 16'th ACM Symposium on Theory of Computing*, April 1984.

[FLP82] M. Fischer, N. Lynch, and M Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Technical Report LCS-TR-282, MIT, 1982.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[Gol86] O. Goldreich. Private communication. 1986.

[GS86] O. Goldreich and L. Shrira. The effect of link failures on computations in asynchronous rings. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, August 1986.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 Copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 Copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 Copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 Copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 Copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 Copy
Dr. G. Hopper, USNR NAVDAC-OOH Department of the Navy Washington, DC 20374	1 Copy

END

JAN.

1988

DTIC

END

JAN.

1988

DTIC